

운영체제 9조 최종 보고서

과목명	운영체제		
교수명	박재근		
학과	전자정보공학부 IT융합전공		
팀장	박준영(20180569)		
팀원	권경현(20182672)	이건(20180587)	최선웅(20192646)

-목차-

I.프로젝트 개요.....

- 주제선정
- 역할 분담

II. 기본 Shell 구현.....

- 명령어 구현
 - mkdir, cd, touch, rmdir

II-1. 메모리 관리.....

- 페이지 할당
- 페이지 해지
- 페이지 fault 처리

II-2. 데이터 구조.....

- 파일 및 디렉토리 메타데이터 구조

III. 결론.....

I.프로젝트 개요

-주제선정

운영체제(OS)는 컴퓨터 시스템의 핵심요소로 그중 파일시스템은 중요한 구성요소입니다. 파일 시스템은 데이터를 저장, 관리, 접근하는 데 필수적인 역할을 하며, 이를 이해하고 구현하는 과정에서 운영체제의 근본적인 작동 원리를 깊게 이해하는 데 도움이 된다고 생각하여 파일 시뮬레이터를 선정 하였습니다.

-구현 환경

1. 개발 도구 및 언어:

- C언어: 운영체제 개발에 널리 사용되는 언어로, 하드웨어와의 직접적인 인터페이스가 가능하며, 메모리 관리 같은 저수준 작업을 효율적으로 수행 가능하게 함. miniOS의 기본 코드가 C언어로 작성 되어있어 일관성을 유지하는데에 적합하기 때문.
- GNU 컴파일러(GCC) : C언어 코드를 컴파일 하는 데 표준적으로 사용되며, 다양한 플랫폼에서 호환성을 보장함.

2.개발 및 테스트 환경

- VirtualBox: 가상 머신 소프트웨어로, 여러 운영체제를 실행 및 테스트 할 수 있는 환경을 제공. miniOS를 다양한 시나리오에서 테스트 가능.

3.소스 코드 관리

- Git : 팀원 간의 협업을 원활하게 하고, 코드 변경 이력을 체계적으로 관리 할 수 있기에 효율적인 능력을 보장.

II. 역할 분담

- 박준영: 메모리 관리 구현 (페이지 할당 및 해지)
- 권경현: 백업 프로그램 구현 (파일 공유)
- 최선웅 : 기본 shell 명령어 구현 (파일 시스템 인터페이스 제공)
- 이건 : 메타데이터 구조 구현 (데이터 요구사항 분석)

* 기본 shell 구현

Shell 인터페이스는 사용자와 파일 시스템 간의 상호작용을 위한 편리한 인터페이스를 제공한다. 사용자가 입력한 명령어를 해석하고 해당 명령어를 실행하는 역할을 하기 위해 기본적인 shell 인터페이스를 구현한다. 또한, 기초적인 실행을 위해 기본적인 명령어 세트를 구현한다.

기본 명령어 구현

- mkdir

- * 현재 디렉토리에 지정된 이름의 새로운 디렉토리를 만든다.
- * mkdir new_folder 의 형태로 명령어를 사용한다.

```
void create_directory(char* directory_name)
{
    char new_directory[MAX_PATH_LENGTH];
    sprintf(new_directory, "%s/%s", current_directory, directory_name);
    int status = mkdir(new_directory, 0777);
    if (status == 0)
    {
        printf("Directory '%s' created successfully\n", new_directory);
    }
    else
    {
        printf("Failed to create directory '%s'\n", new_directory);
    }
}
```

sprintf 함수를 사용해 new_directory에 current_directory와 directory_name을 결합해 새로운 디렉토리의 전체 경로를 만든다.

mkdir 함수를 이용해 새로운 디렉토리를 만든다. 디렉토리 접근 권한을 0777로 설정하여 모든 사용자에게 읽기, 쓰기, 실행 권한을 부여한다.

- cd

- * 지정된 디렉토리로 이동하고, 성공하면 current_directory를 업데이트한다.
- * cd <directory> 의 형태로 명령어를 사용한다.

```
void change_directory(char* directory)
{
    if (chdir(directory) == 0) {
        if (getcwd(current_directory, sizeof(current_directory)) == NULL) {
            perror("Error getting current directory");
        }
    }
    else {
        perror("Error changing directory");
    }
}
```

chdir 함수는 현재 작업 디렉토리를 directory로 변경한다.

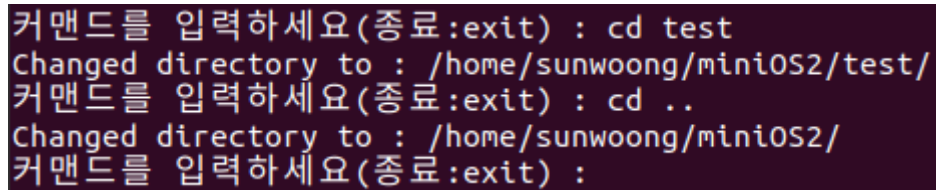
디렉토리 변경에 성공하면 getcwd 함수를 이용해 현재 작업 디렉토리의 경로를 current_directory에 저장한다.

chdir 또는 getcwd 함수가 실패하면 perror 함수를 사용해 적절한 오류 메시지를 출력한다.

중간 보고서에서 발견했던 문제점이 디렉토리의 끝부분에 '/' 가 붙지 않아 발생함을 알아내어 모든 디렉토리 끝에 '/' 가 붙도록 수정해주었다. 또한, 현재의 디렉토리가 어디인지 편하게 볼 수 있도록 관련 print 문을 추가해 주었다.

```
void go_back_directory(){
    if(chdir("..") == 0){
        if (getcwd(current_directory, sizeof(current_directory)) == NULL){
            perror("Error getting current directory");
        }
    } else {
        perror("Error cahnging directory");
    }
}
```

상위 디렉토리로 이동하는 방법을 고민하다 새로운 명령어를 추가해주어 구현해주었다.



```
커맨드를 입력하세요(종료:exit) : cd test
Changed directory to : /home/sunwoong/miniOS2/test/
커맨드를 입력하세요(종료:exit) : cd ..
Changed directory to : /home/sunwoong/miniOS2/
커맨드를 입력하세요(종료:exit) :
```

해당 이미지와 같이 cd 뒤에 .. 이라고 입력되었을 때, 상위 디렉토리로 이동할 수 있도록 해주었다.

- touch

- * 현재 디렉토리에 지정된 이름의 새 파일을 만든다.
- * touch new_file.txt 의 형태로 명령어를 사용한다.

```
void create_file(char* file_name)
{
    char file_path[MAX_PATH_LENGTH];
    snprintf(file_path, sizeof(file_path), "%s%s", current_directory, file_name);

    int fd = open(file_path, O_CREAT | O_EXCL | O_WRONLY, 0644);
    if (fd == -1) {
        perror("Error creating file");
    } else {
        printf("File '%s' created successfully.\n", file_name);
        close(fd);
    }
}
```

snprintf 함수를 사용해 file_path에 current_directory와 file_name을 결합해 새로운 파일의 전체 경로를 만든다.

open 함수를 이용해 파일을 생성하고 열기 위한 옵션을 설정한다. O_CREAT는 파일이 존재하지 않으면 새 파일을 생성해주는 옵션이다. O_EXCL은 O_CREAT와 함께 사용될 때, 파일이 이미 존재하면 실패한다. O_WRONLY는 파일을 쓰기 전용으로 열게 해준다. 파일 접근 권한은 0644로 설정해 소유자에게 읽기 및 쓰기 권한을, 그룹과 다른 사용자에게는 읽기 권한을 부여한다.

```
커맨드를 입력하세요(종료:exit) : cd test
Changed directory to : /home/sunwoong/miniOS2/test/
커맨드를 입력하세요(종료:exit) : touch test_file.txt
File 'test_file.txt' created successfully.
커맨드를 입력하세요(종료:exit) : ls
/home/sunwoong/miniOS2/test/의 내용 :
..
test_file.txt
.
커맨드를 입력하세요(종료:exit) : █
```

중간 보고서 작성 때에는 상위 디렉토리에 만들고자 했던 폴더의 이름이 붙어 testtest_file.txt의 파일이 생성되었지만, cd 명령어를 수정해줌으로써 정상적으로 동작하게 되었다.

- rmdir

```
void remove_directory(char *directory){
    char directory_path[MAX_PATH_LENGTH];
    snprintf(directory_path, sizeof(directory_path), "%s%s", current_directory, directory);

    if(rmdir(directory_path) == 0){
        printf("Directory '%s' removed successfully.\n", directory);
    } else {
        perror("Error removing directory");
    }
}
```

디렉토리를 제거해주는 명령어인 rmdir을 만들었다.

```
커맨드를 입력하세요(종료:exit) : mkdir test
Directory '/home/sunwoong/miniOS2/test' created successfully
커맨드를 입력하세요(종료:exit) : rmdir test
Error removing directory: No such file or directory
커맨드를 입력하세요(종료:exit) : cd test
Changed directory to : /home/sunwoong/miniOS2/test/
커맨드를 입력하세요(종료:exit) : cd ..
Changed directory to : /home/sunwoong/miniOS2/
커맨드를 입력하세요(종료:exit) : rmdir test
Directory 'test' removed successfully.
커맨드를 입력하세요(종료:exit) : █
```

하지만 처음 실행하자마자 제거가 되진 않고, cd 명령어를 이용해 디렉토리를 갱신하여야만 rmdir이 정상적으로 작동함을 볼 수 있다.

* 메모리 관리

페이징을 통해 메모리를 관리하기로 결정했다. 페이지는 동적 할당의 형태로 관리된다. 운영체제는 프로세스에 연속된 가상 주소 공간을 제공하지만, 실제 물리 메모리는 반드시 연속적이진 않을 수 있기 때문에 이를 해결하기 위해 메모리를 고정된 크기의 페이지로 나누어서 관리한다.

메모리 할당 및 해제 알고리즘 구현

- 페이지 할당

* 프로그램이 프로그램이 메모리를 요청할 때, 운영체제는 페이지 테이블을 참조하여 사용 가능한 페이지(프레임)를 찾아 할당, 이 과정에서 필요한 만큼의 연속된 가상 페이지가 물리 프레임에 mapping 된다.

* malloc() 이라는 C 표준 라이브러리에서 제공하는 동적 메모리 할당 함수를 allocate_memory()로 간단하게 구현.

```
void* allocate_memory(int size) {
    int num_pages_needed = (size + PAGE_SIZE - 1) / PAGE_SIZE;
    int start_page = -1;
    int contiguous_pages = 0;

    for (int i = 0; i < NUM_PAGES; i++) {
        if (!memory_manager.pages[i].is_allocated) {
            if (start_page == -1) {
                start_page = i;
            }
            contiguous_pages++;
            if (contiguous_pages == num_pages_needed) {
                break;
            }
        } else {
            start_page = -1;
            contiguous_pages = 0;
        }
    }

    if (contiguous_pages < num_pages_needed) {
        return NULL; // Not enough memory
    }

    for (int i = start_page; i < start_page + num_pages_needed; i++) {
        memory_manager.pages[i].is_allocated = true;
    }

    return (void*)(start_page * PAGE_SIZE);
}
```

역할 : 요청된 크기의 메모리 블록을 할당한다.

동작 원리 : 내부적으로 페이지 테이블을 사용하여 가상 메모리 공간에서 필요한 크기의 연속된 페이지를 찾아 할당한다.

- 페이지 해제

* 프로그램이 더이상 메모리가 필요하지 않거나 종료될 때, 운영체제는 해당 페이지를 해제하고 페이지 테이블을 업데이트하여 다시 사용 가능하도록 한다.

* free() 라는 C 표준 라이브러리에서 제공하는 동적 메모리 해제 함수를 free_memory()로 간단하게 구현.

```
void free_memory(void* ptr, int size) {
    int start_page = (int)ptr / PAGE_SIZE;
    int num_pages_to_free = (size + PAGE_SIZE - 1) / PAGE_SIZE;

    for (int i = start_page; i < start_page + num_pages_to_free; i++) {
        memory_manager.pages[i].is_allocated = false;
    }
}

void print_memory_status() {
    for (int i = 0; i < NUM_PAGES; i++) {
        printf("Page %d: %s\n", i, memory_manager.pages[i].is_allocated ? "Allocated" : "Free");
    }
}
```

역할 : 동적으로 할당된 메모리를 해제한다.

동작 원리 : 페이지 테이블을 업데이트하여 해당 페이지를 비할당 상태로 변경한다.

- 메모리 할당 상태 출력 함수

* print_memory_status()

```
void print_memory_status() {
    for (int i = 0; i < NUM_PAGES; i++) {
        printf("Page %d: %s\n", i, memory_manager.pages[i].is_allocated ? "Allocated" : "Free");
    }
}
```

목적 : 메모리 관리 시스템의 상태를 시각적으로 확인하기 위해 현재 메모리 할당 상태를 출력한다. 디버깅 및 모니터링을 위한 도구로 사용된다.

동작 원리 : 메모리 관리자의 페이지 테이블을 순회하면서 각 페이지의 할당 상태를 확인한다. 각 페이지의 할당 여부를 출력한다.

* "NUM_PAGES" 만큼의 페이지가 있으며, 각 페이지의 할당 상태를 'memory_manager.pages' 배열에서 확인한다.

- 페이지 fault 처리

process가 접근하려는 페이지가 메모리에 존재하지 않을 때, 이를 처리하기 위한 로직이 필요하다.

```
void handle_page_fault(int page_number) {
    printf("Page fault occurred for page %d\n", page_number);

    // Simple page replacement policy: First free page
    for (int i = 0; i < NUM_PAGES; i++) {
        if (!memory_manager.pages[i].is_in_memory) {
            // Simulate loading the page from disk to memory
            memory_manager.pages[i].frame_number = page_number;
            memory_manager.pages[i].is_in_memory = true;
            printf("Loaded page %d into frame %d\n", page_number, i);
            return;
        }
    }

    // 만약에 free memory가 없다면 replacement 알고리즘이 필요하다.
    // 지금 코드는 replacement 알고리즘이 구현되어 있지 않다.
    printf("No free frame available to load page %d\n", page_number);
}
```

```
Page 9: Free (Frame -1)
Page 10: Free (Frame -1)
Page 11: Free (Frame -1)
Page 12: Free (Frame -1)
Page 13: Free (Frame -1)
Page 14: Free (Frame -1)
Page 15: Free (Frame -1)
Accessing page 2 in frame 2
Accessing page 3 in frame 3
Page fault occurred for page 10
Loaded page 10 into frame 0
Accessing page 10 in frame -1
Page fault occurred for page 20
Loaded page 20 into frame 1
Accessing page 20 in frame 0
```

메모리 페이지 2에 접근하려고 시도했고, 이 페이지가 메모리에 없을 경우 페이지 fault가 발생됐다. 페이지 폴트가 발생했을 때, 이를 해결하기 위해서 page replacement 알고리즘이 필요하다.

- 페이지 교체 알고리즘

메모리가 가득 차서 새로운 페이지를 할당할 수 없을 때, 기존의 페이지를 교체해야한다. 이를 위해 FIFO(First In First Out)을 구현하려한다.

- 가상 메모리 지원

프로세스의 가상 주소를 물리 주소로 변환하는 기능 구현.

- TLB (Translation Lookaside Buffer) 구현

TLB는 가상 주소를 물리 주소로 변환하는 속도를 높이기 위한 캐시 메모리이다. 이를 구현하여 페이지 테이블 조회 속도를 개선.

- 메모리 보호

프로세스간의 메모리 접근을 보호하기 위해 페이지에 대한 접근 권한(읽기, 쓰기, 실행)을 설정하고 이를 관리

데이터 구조 역할

프로그램이 효과적으로 작동하도록 하기 위한 데이터 구조를 설계하고 구현하기 위해선 데이터 요구 사항에 대한 구체적인 분석이 필요하다.

1. 파일시뮬레이터 프로그램에서 필요한 데이터 및 그 데이터의 흐름을 분석.
2. 파일 및 디렉터리 구조, 메타데이터, 접근 권한, 파일 읽기/쓰기 작업 등에 대한 요구 사항을 정의한다.
3. 요구 사항에 맞는 최적의 데이터 구조를 선택하고 설계한다.
-> 파일 및 디렉터리 구조를 표현하기 위해 트리 구조를 사용하거나, 파일 내용 관리를 위해 배열 또는 리스트를 사용하는 등 다양한 데이터 구조를 검토해야 한다.

- 본 시스템 구현을 위해 정의된 데이터 구조와 그 구현 예제를 제시해보았다.

1. **파일과 디렉토리 구조체 정의** : 파일과 디렉토리의 구조체를 정의하고, 파일의 메타데이터(이름, 크기, 생성 시간, 수정 시간, 데이터 등)를 포함하도록 설계함.
2. **디렉토리 관리** : 디렉토리에 파일을 추가하고 관리할 수 있는 기능을 구현.
3. **파일 조작 함수** : 파일을 생성하고, 데이터를 쓰고, 읽는 기능을 구현.
4. **시간 관리** : 파일 생성 및 수정 시각을 기록.

제시된 코드에는 파일과 디렉토리 구조체, 파일 생성, 추가, 읽기 및 쓰기 기능이 포함되어 있음. 본 데이터 구조는 miniOS 환경에서 파일 관리 시스템을 시뮬레이션하는 데 사용됨.

파일 메타데이터 구조체 (File)

파일의 이름, 크기, 생성시간, 수정시간, 데이터 등을 저장 하는 구조체이다.

```
// 파일 메타데이터 구조체
typedef struct
{
    char name[MAX_FILENAME];
    size_t size;
    time_t creation_time;
    time_t modification_time;
    char data[MAX_FILESIZE];
} File;
```

디렉토리 구조체 (Directory)

디렉토리 구조체는 디렉토리 이름과 해당 디렉토리에 포함된 파일들의 포인터 배열, 파일 개수를 저장하는 구조체.

```
// 디렉토리 구조체
typedef struct
{
    char name[MAX_FILENAME];
    File *files[MAX_FILES];
    int file_count;
} Directory;
```

루트 디렉토리 생성

루트 디렉토리는 전역 변수로 정의되며, 초기화 시 이름은 "/"로 설정되고, 파일 개수는 0으로 초기화됨.

```
// 루트 디렉토리 생성
Directory root = {"/", {}, 0};
```

파일 생성 및 추가 함수

create_file : 파일 크기만큼 공간을 할당해 생성시간을 저장시킴.

add_file: 디렉토리 공간을 체크한 후 파일을 추가.

```
// 파일 생성 함수
File *create_file(const char *name)
{
    File *new_file = (File *)malloc(sizeof(File));
    if (new_file == NULL)
        return NULL;
    strncpy(new_file->name, name, MAX_FILENAME - 1);
    new_file->name[MAX_FILENAME - 1] = '\0';
    new_file->size = 0;
    new_file->creation_time = time(NULL);
    new_file->modification_time = time(NULL);
    memset(new_file->data, 0, MAX_FILESIZE);
    printf("\n\'%s\' 파일이 생성됨!\n", name);
    return new_file;
}

// 파일 추가 함수
void add_file(Directory *dir, File *file)
{
    if (dir->file_count >= MAX_FILES)
    {
        printf("Directory is full!\n");
        return;
    }
    dir->files[dir->file_count++] = file;
    printf("\n\'%s\' 파일이 디렉토리에 추가 됨!\n\n", *file);
}
```

파일 읽기 및 쓰기 함수

read_file: 파일명, 파일 크기, 생성 및 수정 시간, 파일내의 데이터를 출력하는 함수.

write_file: 파일에 데이터를 삽입시키고 수정 시간을 저장하는 기능.

```
// 파일 읽기 함수
void read_file(File *file)
{
    printf("-----파일 정보-----");
    printf("\n파일: %s\n크기: %lu bytes\n생성시간: %s수정시간: %s데이터: %s\n\n",
        file->name,
        file->size,
        ctime(&file->creation_time),
        ctime(&file->modification_time),
        file->data);
}

// 파일 쓰기 함수
void write_file(File *file, const char *data)
{
    strncpy(file->data, data, MAX_FILESIZE);
    file->data[MAX_FILESIZE - 1] = '\0';
    file->size = strlen(file->data);
    file->modification_time = time(NULL);
}
```

파일 삭제 및 변경 함수

파일의 주소값을 통해 디렉토리에 추가된 파일명을 대조해서 일치하는 파일을 삭제 및 변경 기능

```
// 파일 삭제 함수
void delete_file(Directory *dir, const char *filename)
{
    printf(" \'%s\' 파일 삭제 중..\n", filename);
    for (int i = 0; i < dir->file_count; i++)
    {
        if (strcmp(dir->files[i]->name, filename) == 0)
        {
            free(dir->files[i]);
            for (int j = i; j < dir->file_count - 1; j++)
            {
                dir->files[j] = dir->files[j + 1];
            }
            dir->file_count--;
            printf(" \'%s\' 파일 삭제 완료.\n\n", filename);
            return;
        }
    }

    printf(" \'%s\' 파일을 찾을 수 없음.\n\n", filename);
}

// 파일 이름 변경 함수
void rename_file(File *file, const char *new_name)
{
    strncpy(file->name, new_name, MAX_FILENAME - 1);
    file->name[MAX_FILENAME - 1] = '\0';
}
```

디렉토리 내의 파일 나열 함수

현 디렉토리 내의 전체 파일들을 나열해주는 기능

```
// 디렉토리 내용 나열 함수
void list_directory(Directory *dir)
{
    printf("디렉토리 내용 나열\n");
    printf("Directory: %s\n", dir->name);
    for (int i = 0; i < dir->file_count; i++)
    {
        printf("%d: %s\n", i + 1, dir->files[i]->name);
    }
    printf("\n");
}
```

메인 함수

1. 파일 생성 후 파일에 데이터를 추가 및 파일 정보 출력
2. 또 다른 파일을 생성 후 추가
3. 파일 이름 변경 후 현 디렉토리 내의 파일들 나열
4. 디렉토리 내의 파일 삭제 후 확인 과정

```
int main()
{
    File *file1 = create_file("file1.txt");
    write_file(file1, "Hello, MiniOS!");
    add_file(&root, file1);

    read_file(file1);

    // 새로운 함수 Test
    File *file2 = create_file("file2.txt");
    add_file(&root, file2);
    list_directory(&root);
    rename_file(file2, "new_file2.txt");
    list_directory(&root);
    delete_file(&root, "file1.txt");
    list_directory(&root);

    return 0;
}
```

실행 결과

파일 생성과 디렉토리에 추가 하였을 시 파일 정보가 잘 출력 되는 결과를 볼 수 있고,
파일 삭제 후 디렉토리 내에서 제거가 성공적으로 됐음을 확인 할 수 있음.

```
'file1.txt' 파일이 생성됨!  
'file1.txt' 파일이 디렉토리에 추가 됨!  
  
-----파일 정보-----  
파일 : file1.txt  
크기 : 14 bytes  
생성 시간 : Wed Jun 12 21:27:36 2024  
수정 시간 : Wed Jun 12 21:27:36 2024  
데이터 : Hello, MiniOS!  
  
'file2.txt' 파일이 생성됨!  
'file2.txt' 파일이 디렉토리에 추가 됨!  
  
디렉토리 내용 나열  
Directory: /  
1: file1.txt  
2: file2.txt  
  
디렉토리 내용 나열  
Directory: /  
1: file1.txt  
2: new_file2.txt  
  
'file1.txt' 파일 삭제 중..  
'file1.txt' 파일 삭제 완료.  
  
디렉토리 내용 나열  
Directory: /  
1: new_file2.txt
```

결론

제시된 데이터 구조와 함수들은 miniOS 파일시뮬레이터의 기본적인 파일 및 디렉토리 관리를 구현한다. 이러한 구조는 파일의 생성, 저장, 읽기 및 쓰기와 같은 기본 파일 시스템 기능을 제공하며, 확장성과 유연성을 고려하여 설계되었다. 이를 바탕으로 추후에 더 복잡한 파일 관리 기능을 추가할 수 있다.

파일 공유 프로그램

Main.c

```
else if(strcmp(argv[1],"backup")==0){ //argv[2] file or directory
    if(argc==2){
        fprintf(stderr,"error no path\n");
        exit(1);
    }
    if(argc>=3){
        char realPath[PATHNAME_SIZE]={0,};
        if(realpath(argv[2],realPath)==NULL){
            fprintf(stderr,"convert to realpath fail %s\n",argv[2]);
            exit(1);
        }

        char* timeStr=getTimeStr();
        char* desDirPath;
        if((desDirPath=makeBackupDirPath(timeStr))==NULL){
            fprintf(stderr,"makeBackupDirPath error\n");
            free(timeStr);
            exit(1);
        }
        //////////////////////////////////////
        int result;
        int d=0;
        int r=0;
        int y=0;

        while((c=getopt(argc,argv,"dry"))!=-1){
            switch(c){
                case 'd':
                    d=1;
                    break;
                case 'r':
                    r=1;
                    break;
                case 'y':
                    y=1;
                    break;
                case '?':
                    fprintf(stderr,"option err\n");
                    free(timeStr);
                    free(desDirPath);
                    exit(1);
                    break;
            }
        }

        if(r==1){
            if(y==0)
                result=backup_deep(realPath,desDirPath,1);

            if(y==1)
                result=backup_deep(realPath,desDirPath,0);
        }
        else{
            if(d==1){
                if(y==0)
                    result=backup_shallow(realPath,desDirPath,1);

                if(y==1)
                    result=backup_shallow(realPath,desDirPath,0);
            }
            if(d==0){
                if(y==0)
                    result=backup_file(realPath,desDirPath,1);

                if(y==1)
                    result=backup_file(realPath,desDirPath,0);
            }
        }

        if(result<0){
            fprintf(stderr,"backup error\n");
            free(timeStr);
            free(desDirPath);
            exit(1);
        }
        //////////////////////////////////////
        //realpath, desDirPath, timeStr -> ssubak.log record

        checkDirIsEmpty(desDirPath);
        free(timeStr);
        free(desDirPath);
    }
}
```

흐름 구현 : 명령어와 옵션 입력 -> getopt를 이용한 분기

옵션에 따라 다른 동작 실행

- 1) 단일 파일 공유 : 해당 파일 공유 backup_file
- 2) 단일 폴더 공유 : 해당 폴더 내 파일 다 공유 backup_shallow
- 3) 재귀적인 폴더 공유 : 폴더내 파일 다 공유 backup_deep

구현 backup_file

```
int backup_file(char* srcFilePath, char* desDirPath, int cmpflag){
    if(access(desDirPath,F_OK)<0){
        fprintf(stderr,"desDirPath %s not exist\n",desDirPath);
        return -1;
    }

    if(cmpflag==1){ ... }

    char* filename;
    char desAbPath[PATHNAME_SIZE]={0,};

    if((filename=getFileNameFromAbPath(srcFilePath))==NULL){
        fprintf(stderr,"getFileNameFromAbPath failed %s\n",srcFilePath);
        return -1;
    }

    sprintf(desAbPath,"%s/%s",desDirPath,filename);

    if(copyFile(srcFilePath,desAbPath)<0){
        fprintf(stderr,"copyfile err %s to %s\n",srcFilePath,desAbPath);
        free(filename);
        return -1;
    }

    //copy over finish
    //lets record to file log
    if(writeFileLog(srcFilePath,desDirPath)<0){
        fprintf(stderr,"writeFileLog fail %s\n",filename);
        free(filename);
        return -1;
    }

    free(filename);

    char* timeStr=getFileNameFromAbPath(desDirPath);
    int logfd=initcheck_openbaklog();
    char logbuf[PATHNAME_SIZE]={0,};
    sprintf(logbuf,"%s: %s backed up to %s\n",timeStr,srcFilePath,desAbPath);
    write(logfd,logbuf,PATHNAME_SIZE);
    close(logfd);
    free(timeStr);

    return 0;
}
```

설명 : 파일이 존재 유무 확인 후 절대경로 get. -> 다른 공간으로 복사 동작 -> 해당 파일에 대한 메타 데이터 로그에 기록.

구현 backup_shallow

```
int backup_shallow(char* srcDirPath, char* backupDirPath, int cmpflag){
    DIR* dirp;
    struct dirent* dentry;
    struct stat statbuf;
    char filename[PATHNAME_SIZE]={0,};
    char srcAbPath[PATHNAME_SIZE]={0,};

    if((dirp=opendir(srcDirPath))==NULL){
        fprintf(stderr,"open dir fail %s\n",srcDirPath);
        return -1;
    }

    while((dentry=readdir(dirp))!=NULL){
        if(dentry->d_ino==0)
            continue;

        //filename : relative path
        memcpy(filename,dentry->d_name,PATHNAME_SIZE);

        if(strcmp(filename,".")==0||strcmp(filename,"..")==0)
            continue;

        //rel path => absolute path
        sprintf(srcAbPath,"%s/%s",srcDirPath,filename);

        if(lstat(srcAbPath,&statbuf)==-1){
            fprintf(stderr,"stat error %s\n",srcAbPath);
            closedir(dirp);
            return -1;
        }

        if(S_ISDIR(statbuf.st_mode)){
            continue;//different at backup_deep();
        }
        else if(S_ISREG(statbuf.st_mode)){
            if(backup_file(srcAbPath,backupDirPath,cmpflag)<0){
                fprintf(stderr,"backup_file fail %s\n",srcAbPath);
            }
        }
    }

    closedir(dirp);
    return 0;
}
```

설명 : 해당 디렉토리 존재 유무 확인 후 절대경로 get -> 디렉토리 내부 파일들 순회 하며 내부 디렉토리는 무시.
파일은 backup_file함수 호출하여 동작. 재귀적인 동작 X

구현 backup_deep

```
//success : ret 0, fail : ret -1
//no recursive
int backup_deep(char* srcDirPath, char* backupDirPath, int cmpflag){
    DIR* dirp;
    struct dirent* dentry;
    struct stat statbuf;
    char filename[PATHNAME_SIZE]={0,};
    char srcAbPath[PATHNAME_SIZE]={0,};

    NAME_QUEUE queue;
    name_queueInit(&queue);
    enqueue(&queue,srcDirPath);
    char* remainDirPath;

    NAME_QUEUE backupQueue;
    name_queueInit(&backupQueue);
    enqueue(&backupQueue,backupDirPath);
}
```

```

while((remainDirPath=dename_queue(&queue))!=NULL){
    char* tempBackupDirPath=dename_queue(&backupQueue);

    if((dirp=opendir(remainDirPath))==NULL){
        fprintf(stderr,"open dir fail %s\n",remainDirPath);
        free(tempBackupDirPath);
        free(remainDirPath);
        clearname_queue(&queue);
        return -1;
    }

    while((dentry=readdir(dirp))!=NULL){
        if(dentry->d_ino==0)
            continue;

        //filename : relative path
        memcpy(filename,dentry->d_name,PATHNAME_SIZE);

        if(strcmp(filename,".")==0||strcmp(filename,"..")==0)
            continue;

        //rel path => absolute path
        sprintf(srcAbPath,"%s/%s",remainDirPath,filename);

        if(lstat(srcAbPath,&statbuf)==-1){
            fprintf(stderr,"stat error %s\n",srcAbPath);
            free(tempBackupDirPath);
            free(remainDirPath);
            closedir(dirp);
            clearname_queue(&queue);
            return -1;
        }

        if(S_ISDIR(statbuf.st_mode)){
            enname_queue(&queue,srcAbPath);
            char tempPath[PATHNAME_SIZE]={0,};
            sprintf(tempPath,"%s/%s",backupDirPath,filename);
            enname_queue(&backupQueue,tempPath);
        }
        else if(S_ISREG(statbuf.st_mode)){
            if(access(tempBackupDirPath,F_OK)<0){
                mkdir(tempBackupDirPath,0777);
            }

            if(backup_file(srcAbPath,tempBackupDirPath,cmpflag)<0){
                fprintf(stderr,"backup_file fail %s\n",srcAbPath);
            }
        }
    }
    free(tempBackupDirPath);
    closedir(dirp);
    free(remainDirPath);
}

```

설명 : 해당 디렉토리 존재 유무 확인 후 절대경로 get -> 디렉토리 내부 파일들 순회 하며 내부 queue에. 파일은 backup_file함수 호출하여 동작. 파일 모두 순회 후 queue에서 하나 dequeue -> dequeue한 폴더 기준 재귀적인 동작 반복.

추가적인 고려사항

다른 외부 ip의 기기에게 전달을 하려면 해당 기기의 내부 네트워크의 ip와 내부 포트번호를 알아야하는데 이는 수신측에서 공인 ip와 port넘버를 연결해야하는 포트포워딩 동작이 필요함.

파일 공유 범위를 내부 네트워크로 할지, 외부까지 넓힐지는 좀더 고민해봐야한다.

-Project 결론

miniOS를 이용한 파일 시뮬레이터 프로젝트는 운영체제의 기본 개념과 파일 시스템의 원리를 이해하고, 이를 실제로 구현해보았다.

기본 Shell 명령어 구현, 메모리 관리, 파일 메타데이터 분석, 파일 공유 및 백업 기능을 구현 하였다. 기본 shell 명령어는 사용자와 시스템 간의 효율적인 상호작용을 가능하게 하였으며, 메모리 관리 시스템은 자원의 효율적인 분배와 사용을 보장하였다. 파일 메타데이터 구현을 통해 파일의 속성과 상태를 체계적으로 관리할 수 있었고, 파일 백업 및 공유 기능은 데이터의 안정성과 접근성을 높였다.

본 프로젝트를 통해 miniOS와 같은 소형 운영체제를 활용하여 복잡한 시스템의 기본 구조를 학습 하였고, 운영체제의 다양한 구성 요소들이 어떻게 상호작용하는지 명확히 파악 할 수 있었다.